



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих
комп'ютерних систем**

Лабораторна робота №3

з дисципліни **Бази даних і засоби управління**
на тему: “Засоби оптимізації роботи СУБД PostgreSQL”

Виконала:
студент III курсу
групи КВ-94
Холодар А. А.
Перевірив:
Петрашенко А. В.

Київ – 2021

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

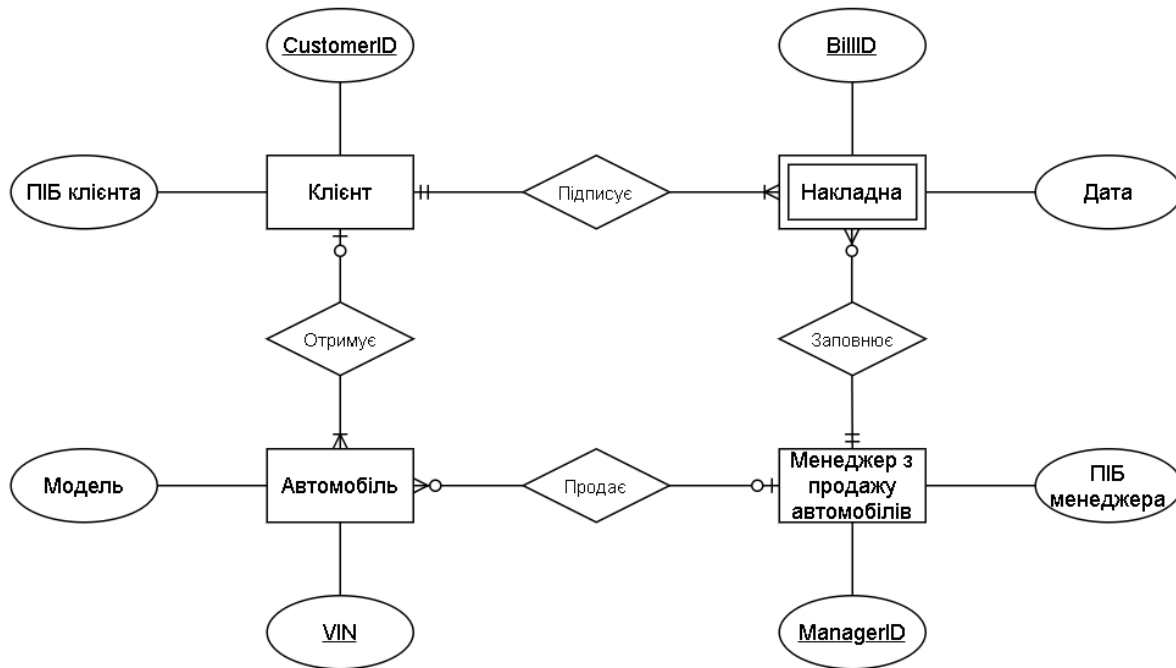
Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

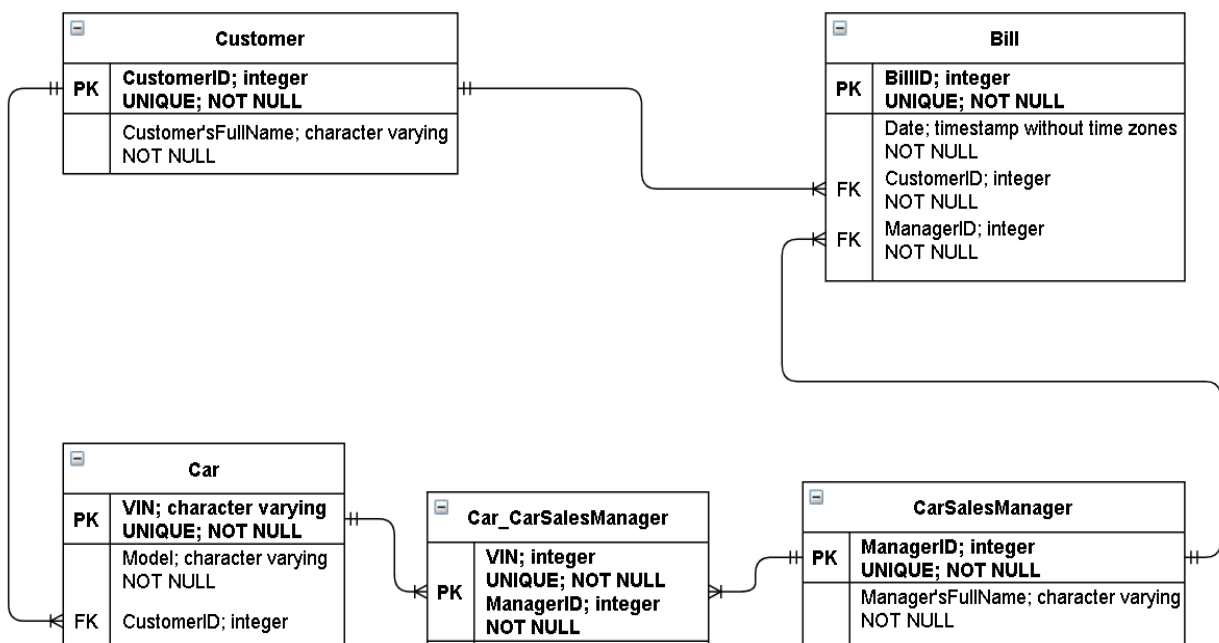
24	<i>GIN, BRIN</i>	<i>after update, insert</i>
----	------------------	-----------------------------

Завдання 1

Модель «сутність-зв'язок» галузі продажу автомобілей автосалоном



Перетворення моделі у схему бази даних



Таблиці бази даних у pgAdmin 4

-- Table: public.Customer

-- DROP TABLE public."Customer";

```
CREATE TABLE IF NOT EXISTS public."Customer"
(
    "CustomerID"          integer          NOT          NULL          DEFAULT
nextval("Customer_CustomerID_seq"::regclass),
    "CustomersFullName" character varying COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "Customer_pkey" PRIMARY KEY ("CustomerID")
)
```

TABLESPACE pg_default;

```
ALTER TABLE public."Customer"
    OWNER to postgres;
```

-- Table: public.Bill

-- DROP TABLE public."Bill";

```
CREATE TABLE IF NOT EXISTS public."Bill"
(
    "BillID" integer NOT NULL DEFAULT nextval("Bill_BillID_seq"::regclass),
    "Date" timestamp without time zone NOT NULL,
    "CustomerID" integer NOT NULL,
    "ManagerID" integer NOT NULL,
    CONSTRAINT "Bill_pkey" PRIMARY KEY ("BillID"),
    CONSTRAINT "Bill_CustomerID_fkey" FOREIGN KEY ("CustomerID")
        REFERENCES public."Customer" ("CustomerID") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT "Bill_ManagerID_fkey" FOREIGN KEY ("ManagerID")
        REFERENCES public."CarSalesManager" ("ManagerID") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

```
TABLESPACE pg_default;
```

```
ALTER TABLE public."Bill"  
    OWNER to postgres;
```

```
-----  
-- Table: public.Car
```

```
-- DROP TABLE public."Car";
```

```
CREATE TABLE IF NOT EXISTS public."Car"  
(  
    "VIN" character varying COLLATE pg_catalog."default" NOT NULL,  
    "Model" character varying COLLATE pg_catalog."default" NOT NULL,  
    "CustomerID" integer NOT NULL,  
    CONSTRAINT "Car_pkey" PRIMARY KEY ("VIN"),  
    CONSTRAINT "Car_CustomerID_fkey" FOREIGN KEY ("CustomerID")  
        REFERENCES public."Customer" ("CustomerID") MATCH SIMPLE  
        ON UPDATE NO ACTION  
        ON DELETE NO ACTION  
)
```

```
TABLESPACE pg_default;
```

```
ALTER TABLE public."Car"  
    OWNER to postgres;
```

```
-----  
-- Table: public.CarSalesManager
```

```
-- DROP TABLE public."CarSalesManager";
```

```
CREATE TABLE IF NOT EXISTS public."CarSalesManager"  
(  
    "ManagerID"          integer          NOT          NULL          DEFAULT  
nextval("CarSalesManager_ManagerID_seq"::regclass),  
    "ManagersFullName" character varying COLLATE pg_catalog."default" NOT NULL,  
    CONSTRAINT "CarSalesManager_pkey" PRIMARY KEY ("ManagerID")  
)
```

TABLESPACE pg_default;

ALTER TABLE public."CarSalesManager"

OWNER to postgres;

-- Table: public.Car_CarSalesManager

-- DROP TABLE public."Car_CarSalesManager";

CREATE TABLE IF NOT EXISTS public."Car_CarSalesManager"

(

"SaleID" integer NOT NULL DEFAULT

nextval('Car_CarSalesManager_SaleID_seq'::regclass),

"VIN" character varying COLLATE pg_catalog."default" NOT NULL,

"ManagerID" integer NOT NULL,

CONSTRAINT "Car_CarSalesManager_pkey" PRIMARY KEY ("SaleID"),

CONSTRAINT "Car_CarSalesManager_ManagerID_fkey" FOREIGN KEY ("ManagerID")

REFERENCES public."CarSalesManager" ("ManagerID") MATCH SIMPLE

ON UPDATE NO ACTION

ON DELETE NO ACTION,

CONSTRAINT "Car_CarSalesManager_VIN_fkey" FOREIGN KEY ("VIN")

REFERENCES public."Car" ("VIN") MATCH SIMPLE

ON UPDATE NO ACTION

ON DELETE NO ACTION

)

TABLESPACE pg_default;

ALTER TABLE public."Car_CarSalesManager"

OWNER to postgres;

Реалізовані класи ORM

```
class Customer(Base):
    __tablename__ = 'Customer'
    CustomerID = Column(INTEGER, primary_key=True, nullable=False,
unique=True)
    CustomersFullName = Column(VARCHAR, nullable=False)
    Bill = relationship("Bill")
    Car = relationship("Car")

    def __init__(self, CustomerID, CustomersFullName):
        self.CustomerID = CustomerID
        self.CustomersFullName = CustomersFullName

    def __repr__(self):
        return "<Customer(CustomerID = {}, CustomersFullName =
'{}')>".format(self.CustomerID, self.CustomersFullName)
-----

class Bill(Base):
    __tablename__ = 'Bill'
    BillID = Column(INTEGER, primary_key=True, nullable=False, unique=True)
    Date = Column(TIMESTAMP, nullable=False)
    CustomerID = Column(INTEGER, ForeignKey('Customer.CustomerID'),
nullable=False)
    ManagerID = Column(INTEGER, ForeignKey('CarSalesManager.ManagerID'),
nullable=False)

    def __init__(self, BillID, Date, CustomerID, ManagerID):
        self.BillID = BillID
        self.Date = Date
        self.CustomerID = CustomerID
        self.ManagerID = ManagerID

    def __repr__(self):
        return "<Bill(BillID = {}, Date = '{}', CustomerID = {}, ManagerID =
{})>\n".format(self.BillID, self.Date, self.CustomerID, self.ManagerID)
-----

class Car(Base):
    __tablename__ = 'Car'
    VIN = Column(VARCHAR, primary_key=True, nullable=False, unique=True)
    Model = Column(VARCHAR, nullable=False)
    CustomerID = Column(INTEGER, ForeignKey('Customer.CustomerID'),
nullable=False)
    Car_CarSalesManager = relationship("Car_CarSalesManager")

    def __init__(self, VIN, Model, CustomerID):
        self.VIN = VIN
        self.Model = Model
        self.CustomerID = CustomerID

    def __repr__(self):
        return "<Car(VIN = '{}', Model = '{}', CustomerID = {})>" \
            .format(self.VIN, self.Model, self.CustomerID)
-----

class CarSalesManager(Base):
    __tablename__ = 'CarSalesManager'
    ManagerID = Column(INTEGER, primary_key=True, nullable=False,
unique=True)
    ManagersFullName = Column(VARCHAR, nullable=False)
    Bill = relationship("Bill")
    Car_CarSalesManager = relationship("Car_CarSalesManager")
```

```

def __init__(self, ManagerID, ManagersFullName):
    self.ManagerID = ManagerID
    self.ManagersFullName = ManagersFullName

def __repr__(self):
    return "<CarSalesManager(ManagerID = {}, ManagersFullName =
'{}')>".format(self.ManagerID, self.ManagersFullName)
-----

class Car_CarSalesManager(Base):
    __tablename__ = 'Car_CarSalesManager'
    SaleID = Column(INTEGER, primary_key=True, nullable=False, unique=True)
    VIN = Column(VARCHAR, ForeignKey('Car.VIN'), nullable=False)
    ManagerID = Column(INTEGER, ForeignKey('CarSalesManager.ManagerID'),
nullable=False)

    def __init__(self, SaleID, VIN, ManagerID):
        self.SaleID = SaleID
        self.VIN = VIN
        self.ManagerID = ManagerID

    def __repr__(self):
        return "<Car_CarSalesManager(SaleID = {}, VIN = '{}', ManagerID =
{})>".format(self.SaleID, self.VIN, self.ManagerID)
-----

```


Запити до таблиць, реалізованих за допомогою ORM

Insert

(на прикладі відношення “Customer” -> “Bill”)

Початковий вигляд таблиць:

```
<Customer(CustomerID = 1, CustomersFullName = 'Just God')>
```

```
<Bill(BillID = 1, Date = '2020-04-04 00:00:00', CustomerID = 1, ManagerID = 1)>
```

Запис до таблиці “Customer”:

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

```
Please, input table number: 1
```

```
Input CustomerID: 2
```

```
Input Customer'sFullName: Genius Prostous
```

```
inserted
```

```
<Customer(CustomerID = 1, CustomersFullName = 'Just God')>
```

```
<Customer(CustomerID = 2, CustomersFullName = 'Genius Prostous')>
```

Спроба запису до таблиці, рядка з первинним ключем, який вже там знаходиться:

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

```
Please, input table number: 1
```

```
Input CustomerID: 2
```

```
Input Customer'sFullName: Just User
```

```
CustomerID = 2 is exist.
```

Запис до таблиці “Bill”:

- 1 -> Customer
- 2 -> Bill
- 3 -> Car
- 4 -> CarSalesManager
- 5 -> Car_CarSalesManager

Please, input table number: 2

Input BillID: 2

Input Date: 2020-05-21

Input CustomerID: 1

Input ManagerID: 1

inserted

<Bill(BillID = 1, Date = '2020-04-04 00:00:00', CustomerID = 1, ManagerID = 1)>

<Bill(BillID = 2, Date = '2020-05-21 00:00:00', CustomerID = 1, ManagerID = 1)>

Спроба запису до таблиці рядка з вторинним ключем, який не відповідає первинному таблиці “Customer” та спроба запису до таблиці рядка з первинним ключем, який вже існує:

- 1 -> Customer
- 2 -> Bill
- 3 -> Car
- 4 -> CarSalesManager
- 5 -> Car_CarSalesManager

Please, input table number: 2

Input BillID: 2

Input Date: 2020-09-09

Input CustomerID: 1

Input ManagerID: 1

BillID = 2 is exist or CustomerID = 1 is not exist or ManagerID = 1 is not exist.

1 -> Continue insertion in this table

2 -> Stop insertion in this table

Your choice -> 1

Input BillID: 3

Input Date: 2020-09-09

Input CustomerID: 29

Input ManagerID: 1

BillID = 3 is exist or CustomerID = 29 is not exist or ManagerID = 1 is not exist.

Delete

(на прикладі відношення “Car” -> “Car_CarSalesManager”)

Початковий вигляд таблиць:

```
<Car(VIN = 'fasdhadfhajdf', Model = 'Skoda', CustomerID = 1)>
<Car(VIN = 'LKJFDH', Model = 'Porsche', CustomerID = 1)>

<Car_CarSalesManager(SaleID = 1, VIN = 'fasdhadfhajdf', ManagerID = 1)>
```

Видалення рядка, у якого нема залежності первинного ключа у інших таблицях:

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

Please, input table number: 3

Input VIN: LKJFDH

deleted

```
<Car(VIN = 'fasdhadfhajdf', Model = 'Skoda', CustomerID = 1)>
```

Спроба видалення рядка, у якого є залежності, або якого не існує:

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

Please, input table number: 3

Input VIN: fasdhadfhajdf

There are some dependencies in Car_CarSalesManager on it
Try something another.

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

Please, input table number: 3

Input VIN: PROMUA

Nothing to delete.

Try something another.

Update

(на прикладі відношення таблиці “CarSalesManager”)

Початковий вигляд таблиці:

```
<CarSalesManager(ManagerID = 1, ManagersFullName = 'Just Error')>
```

Спроба редагування:

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

Please, input table number: 4

Input ManagerID: 1

Input Manager'sFullName: *Poroshenko Petro*

updated

```
<CarSalesManager(ManagerID = 1, ManagersFullName = 'Poroshenko Petro')>
```

Спроба редагування рядка, якого не існує:

```
1 -> Customer
2 -> Bill
3 -> Car
4 -> CarSalesManager
5 -> Car_CarSalesManager
```

Please, input table number: 4

Input ManagerID: 2

Input Manager'sFullName: *Proger3000*

ManagerID = 2 is not exist

Завдання 2

Для тестування індексів було створено окремі таблиці у базі даних test з 1000000 записів.

GIN

GIN – так званий обернений індекс. Він працює з типами даних, значення яких не є атомарними, а складаються з елементів. При цьому індексуються не самі значення, а окремі елементи; кожен елемент посилається на ті значення, у яких він зустрічається. Індекс GIN зберігає набір пар виду: ключ, список появи ключа – де список появи — набір ідентифікаторів рядків, у яких міститься ключ. Один і той самий ідентифікатор рядка може знаходитись у кількох списках. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже швидкий для випадків, коли один і той же ключ з'являється багато разів.

Запити мовою SQL

```
DROP TABLE IF EXISTS "GIN_test";

CREATE TABLE "GIN_test"("id" bigserial PRIMARY KEY, "doc" text, "doc_tsv" tsvector);

INSERT INTO "GIN_test"("doc") SELECT chr(trunc(65 + random()*25)::int)||chr(trunc(65 +
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 +
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 +
random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 + random()*25)::int)||chr(trunc(65 +
random()*25)::int)||chr(trunc(65 + random()*25)::int) FROM generate_series(1,
1000000) as q;

UPDATE "GIN_test" SET "doc_tsv" = to_tsvector("doc");

-----

SELECT COUNT(*) FROM "GIN_test" where "id" % 11 = 0;
SELECT COUNT(*) FROM "GIN_test" WHERE ("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
SELECT AVG("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYBP')) or
("doc_tsv" @@ to_tsquery('RQFWRFPGUJYIF'));
SELECT MAX("id") from "GIN_test" where ("doc_tsv" @@ to_tsquery('WBHWIWGEFCYBP')) GROUP
BY "id" % 11 = 0;

-----

DROP INDEX IF EXISTS "GIN_time_index";
CREATE INDEX "GIN_time_index" ON "GIN_test" USING gin("doc_tsv");
```


З результатів бачимо, що використання індексування GIN значно підвищило швидкість пошуку даних (окрім першого запиту, тому що на даний виклик індексування не впливає). В цілому, така поведінка є очікуваною, тому що основна ідея індексування GIN це те, що кожне значення шуканого ключа зберігається лише один раз і запит йде лише по тим даним, що містяться у списку появи цього ключа.

BRIN

BRIN – техніка індексації даних, призначена для обробки великих таблиць, в яких значення індексованого стовпця має деяку природну кореляцію з фізичним положенням рядка в таблиці. Вони мають такі якості партиціонованих таблиць, як швидка вставка рядка, швидке створення індексу, без необхідності явного оголошення партицій. Спрощено кажучи, BRIN добре працює для тих стовпців, значення яких корелюють з їх фізичним розташуванням у таблиці.

Працює це наступним чином. Таблиця розбивається на зони (range) розміром кілька сторінок (або блоків, що те саме) — звідси й назва: Block Range Index, BRIN. Для кожної зони в індексі зберігається інформація про дані в цій зоні. Як правило, це мінімальне та максимальне значення, але буває інакше. Якщо при виконанні запиту, що містить умову на стовпець, шукані значення не потрапляють у діапазон, всю зону можна сміливо пропускати; якщо ж потрапляють - усі рядки у всіх блоках зони доведеться переглянути та вибрати серед них підходящі.

Запити мовою SQL

```
DROP TABLE IF EXISTS "BRIN_test";  
CREATE TABLE "BRIN_test"("id" int PRIMARY KEY, "date" timestamp NOT NULL, "level" integer,  
"msg" text);  
INSERT INTO "BRIN_test"("id", "date", "level", "msg") SELECT q, CURRENT_TIMESTAMP + ( q ||  
'minute' ) :: interval, random() * 6, md5(q::text) FROM generate_series(1,1000000) as q;
```

```
SELECT COUNT(*) FROM "BRIN_test" where "date" between '2021-12-26 18:35:10.318196' and  
'2021-12-27 02:03:10.318196';
```

```
SELECT MAX("date") FROM "BRIN_test" where "date" between '2021-12-27 00:33:10.318196' and  
'2021-12-27 02:45:10.318196';  
SELECT COUNT(*) FROM "BRIN_test" where "date" <= '2021-12-26 18:35:10.318196';  
SELECT AVG("id") FROM "BRIN_test" where "date" >= '2023-10-30 10:51:10.318196';
```

```
DROP INDEX IF EXISTS "BRIN_test_index";  
CREATE INDEX "BRIN_test_index" ON "BRIN_test" USING brin("date");
```

Перевірка результатів

До індексування:

```
SQL Shell (psql)  
Секундомер включён.  
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" between '2021-12-26 18:35:10.318196' and '2021-12-27 02:03:10.318196';  
count  
-----  
449  
(1 ёёёёёёёё)  
  
Время: 136,244 мс  
test=# SELECT MAX("date") FROM "BRIN_test" where "date" between '2021-12-27 00:33:10.318196' and '2021-12-27 02:45:10.318196';  
max  
-----  
2021-12-27 02:45:10.318196  
(1 ёёёёёёёё)  
  
Время: 139,982 мс  
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" <= '2021-12-26 18:35:10.318196';  
count  
-----  
211  
(1 ёёёёёёёё)  
  
Время: 124,363 мс  
test=# SELECT AVG("id") FROM "BRIN_test" where "date" >= '2023-10-30 10:51:10.318196';  
avg  
-----  
984433.500000000000  
(1 ёёёёёёёё)  
  
Время: 123,504 мс  
test=#
```


Після індексування:

```
SQL Shell (psql)
```

```
test=# CREATE INDEX "BRIN_test_index" ON "BRIN_test" USING brin("date");
CREATE INDEX
Время: 177,393 мс
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" between '2021-12-26 18:35:10.318196' and '2021-12-27 02:03:10.318196';
count
-----
      449
(1 строка)

Время: 3,100 мс
test=# SELECT MAX("date") FROM "BRIN_test" where "date" between '2021-12-27 00:33:10.318196' and '2021-12-27 02:45:10.318196';
max
-----
2021-12-27 02:45:10.318196
(1 строка)

Время: 1,965 мс
test=# SELECT COUNT(*) FROM "BRIN_test" where "date" <= '2021-12-26 18:35:10.318196';
count
-----
      211
(1 строка)

Время: 1,824 мс
test=# SELECT AVG("id") FROM "BRIN_test" where "date" >= '2023-10-30 10:51:10.318196';
avg
-----
984433.500000000000
(1 строка)

Время: 4,081 мс
test=#
```

З результатів, отриманих до і після використання індексування BRIN бачимо, що швидкість пошуку необхідних даних значно збільшилася. Знову ж таки, така поведінка є очікуваною, тому що індекси створені для пришвидшення пошуку необхідної інформації. Індеси BRIN ефективні, якщо впорядкування значень ключів відповідає організації блоків на рівні зберігання. У найпростішому випадку це може вимагати фізичного впорядкування таблиці, яке часто є порядком створення рядків у ній, щоб відповідати порядку ключа. Ключі до згенерованих порядкових номерів або створених даних є найкращими кандидатами для індексу BRIN.

Завдання 3

Заняти мовою SQL:

```
DROP TABLE IF EXISTS "trigger_test";
CREATE TABLE "trigger_test"("trigger_id" bigserial PRIMARY KEY, "trigger_text" text);
DROP TABLE IF EXISTS "test";
CREATE TABLE "test"("test_id" bigserial primary key, "count" bigint, "test_text" text);
```

```
INSERT INTO "trigger_test"("trigger_text") SELECT md5(q::text) FROM generate_series(1,100) as q;
```

```
CREATE OR REPLACE FUNCTION after_update_insert_func()
RETURNS trigger
AS $$
DECLARE
    CURSOR_LOG CURSOR FOR SELECT * FROM "trigger_test";
    row_ "trigger_test"%ROWTYPE;
BEGIN
    IF new."trigger_id" IS NOT NULL THEN
        IF new."trigger_id" % 11 = 0 then
            FOR row_ IN CURSOR_LOG LOOP
                DELETE FROM "test" where "test_id" = 789;
            END LOOP;
            INSERT INTO "test"("count", "test_text") values (new."trigger_id", new."trigger_text");
            RETURN NEW;
        ELSE
            FOR row_ IN CURSOR_LOG LOOP
                DELETE FROM "test" where "test_id" = 456;
            END LOOP;
            INSERT INTO "test"("count", "test_text") values (new."trigger_id", 'OctaviuS');
            RETURN NEW;
        END IF;
    ELSE
        RAISE NOTICE 'old value is null';
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE PLPGSQL;
```

```
CREATE TRIGGER "after_update_insert_trigger"
AFTER UPDATE OR INSERT ON "trigger_test"
FOR EACH ROW
EXECUTE PROCEDURE after_update_insert_func();
```

```
INSERT INTO "trigger_test"("trigger_id", "trigger_text") VALUES (2697, 'STUDENT OF KPI SINCE
2019');
UPDATE "trigger_test" SET "trigger_text" = 'yuppy' WHERE "trigger_id" <= 50;
```

Результати:

Початковий стан таблиць:

“trigger_test”:

Data Output			
	trigger_id [PK] bigint		trigger_text text
33	33		182be0c5cdcd5072bb1864cdee4d3d6e
34	34		e369853df766fa44e1ed0ff613f563bd
35	35		1c383cd30b7c298ab50293adfecb7b18
36	36		19ca14e7ea6328a42e0eb13d585e4c22
37	37		a5bfc9e07964f8dddeb95fc584cd965d
38	38		a5771bce93e200c36f7cd9dfd0e5deaa
39	39		d67d8ab4f4c10bf22aa353e27879133c
40	40		d645920e395fedad7bbbed0eca3fe2e0
41	41		3416a75f4cea9109507cacd8e2f2aefc
42	42		a1d0c6e83f027327d8461063f4ac58a6
43	43		17e62166fc8586dfa4d1bc0e1742c08b
44	44		f7177163c833dff4b38fc8d2872f1ec6
45	45		6c8349cc7260ae62e3b1396831a8398f
46	46		d9d4f495e875a2e075a1a4a6e1b9770f
47	47		67c6a1e7ce56d3d6fa748ab6d9af3fd7
Query Editor Query History Data Output Explain			

“test”:

Data Output

	test_id [PK] bigint	count bigint	test_text text

Query Editor Query History Data Output

Таблиці, після окремого виконання запиту на редагування:

UPDATE "trigger_test" SET "trigger_text" = 'yuppy' WHERE "trigger_id" <= 50;

“trigger_test”:

Data Output

	trigger_id [PK] bigint	trigger_text text
41	41	yuppy
42	42	yuppy
43	43	yuppy
44	44	yuppy
45	45	yuppy
46	46	yuppy
47	47	yuppy
48	48	yuppy
49	49	yuppy
50	50	yuppy
51	51	2838023a778dfaecd212708f721b788
52	52	9a1158154dfa42caddbd0694a4e9bdc8
53	53	d82c8d1619ad8176d665453cfb2e55f0
54	54	a684ecccc76fc522773286a895bc8436
55	55	b53b3a3d6ab90ce0268229151c9bde11

Query Editor

Query History

Data Output

Explain

“test”:

Data Output			
	test_id [PK] bigint	count bigint	test_text text
9	9	9	OctaviuS
10	10	10	OctaviuS
11	11	11	yuppy
12	12	12	OctaviuS
13	13	13	OctaviuS
14	14	14	OctaviuS
15	15	15	OctaviuS
16	16	16	OctaviuS
17	17	17	OctaviuS
18	18	18	OctaviuS
19	19	19	OctaviuS
20	20	20	OctaviuS
21	21	21	OctaviuS
22	22	22	yuppy
23	23	23	OctaviuS

Query Editor Query History **Data Output** Explain

З таблиць бачимо, що після відповідного виконання команди редагування, у таблиці “trigger_test” редагувалися відповідним чином перші 50 рядків, після чого, посилаючись на нові дані, заповнилася таблиця “test”.

Таблиці, після окремого виконання запиту на вставку:

INSERT INTO "trigger_test"("trigger_id", "trigger_text") VALUES (2697, 'STUDENT OF KPI SINCE 2019');

“trigger_test”:

Data Output

	trigger_id [PK] bigint	trigger_text text
1	1	c4ca4238a0b923820dcc509a6f75849b
2	2	c81e728d9d4c2f636f067f89cc14862c
3	3	eccbc87e4b5ce2fe28308fd9f2a7baf3
4	4	a87ff679a2f3e71d9181a67b7542122c
5	5	e4da3b7fbfce2345d7772b0674a318d5
6	6	1679091c5a880faf6fb5e6087eb1b2dc
7	7	8f14e45fceeaa167a5a36dedd4bea2543
8	8	c9f0f895fb98ab9159f51fd0297e236d
9	9	45c48cce2e2d7fbdea1afc51c7c6ad26
10	10	d3d9446802a44259755d38e6d163e820
11	11	6512bd43d9caa6e02c990b0a82652dca
12	12	c20ad4d76fe97759aa27a0c99bfff6710
13	13	c51ce410c124a10e0db5e4b97fc2af39
14	14	aab3238922bcc25a6f606eb525ffdc56
15	15	9bf31c7ff062936a96d3c8bd1f8f2ff3

87	87	c7e1249ffc03eb9ded908c236bd1996d
88	88	2a38a4a9316c49e5a833517c45d31070
89	89	7647966b7343c29048673252e490f736
90	90	8613985ec49eb8f757ae6439e879bb2a
91	91	54229abfcfa5649e7003b83dd4755294
92	92	92cc227532d17e56e07902b254dfad10
93	93	98dce83da57b0395e163467c9dae521b
94	94	f4b9ec30ad9f68f89b29639786cb62ef
95	95	812b4ba287f5ee0bc9d43bbf5bbe87fb
96	96	26657d5ff9020d2abefe558796b99584
97	97	e2ef524fbf3d9fe611d5a8e90fefdc9c
98	98	ed3d2c21991e3bef5e069713af9fa6ca
99	99	ac627ab1ccbdb62ec96e702f07f6425b
100	100	f899139df5e1059396431415e770c6dd
101	2697	STUDENT OF KPI SINCE 2019

Query Editor

Query History

Data Output

Explain

“test”:

Data Output			
	test_id [PK] bigint	count bigint	test_text text
1	1	2697	Octavius

Query Editor Query History **Data Output** Explain

З результатів, отриманих після окремого використання запиту на вставку, бачимо, що як і у попередньому прикладі, спочатку було реалізовано зміни з таблицею “trigger_test”, а вже потім необхідні змінені дані було занесено до таблиці “test”. Дана поведінка є очікуваною.

Таблиці, після виконання обох запитів один за одним:

INSERT INTO "trigger_test"("trigger_id", "trigger_text") VALUES (2697, 'STUDENT OF KPI SINCE 2019');

UPDATE "trigger_test" SET "trigger_text" = 'yuppy' WHERE "trigger_id" <= 50;

“trigger_test”:

	trigger_id [PK] bigint	trigger_text text
38	38	yuppy
39	39	yuppy
40	40	yuppy
41	41	yuppy
42	42	yuppy
43	43	yuppy
44	44	yuppy
45	45	yuppy
46	46	yuppy
47	47	yuppy
48	48	yuppy
49	49	yuppy
50	50	yuppy
51	51	2838023a778dfaecd212708f721b788
52	52	9a1158154dfa42caddbd0694a4e9bdc8
53	53	d82c8d1619ad8176d665453cfb2e55f0
54	54	a684ecccc76fc522773286a895bc8436
55	55	b53b3a3d6ab90ce0268229151c9bde11
56	56	9f61408e3afb633e50cdf1b20de6f466
83	83	fe9fc289c3ff0af142b6d3bead98a923
84	84	68d30a9594728bc39aa24be94b319d21
85	85	3ef815416f775098fe977004015c6193
86	86	93db85ed909c13838ff95ccfa94cebd9
87	87	c7e1249ffc03eb9ded908c236bd1996d
88	88	2a38a4a9316c49e5a833517c45d31070
89	89	7647966b7343c29048673252e490f736
90	90	8613985ec49eb8f757ae6439e879bb2a
91	91	54229abfcfa5649e7003b83dd4755294
92	92	92cc227532d17e56e07902b254dfad10
93	93	98dce83da57b0395e163467c9dae521b
94	94	f4b9ec30ad9f68f89b29639786cb62ef
95	95	812b4ba287f5ee0bc9d43bbf5bbe87fb
96	96	26657d5ff9020d2abefe558796b99584
97	97	e2ef524fbf3d9fe611d5a8e90fefd9c
98	98	ed3d2c21991e3bef5e069713af9fa6ca
99	99	ac627ab1ccbdb62ec96e702f07f6425b
100	100	f899139df5e1059396431415e770c6dd
101	2697	STUDENT OF KPI SINCE 2019

“test”:

Data Output

	test_id [PK] bigint	count bigint	test_text text
1	1	2697	OctaviuS
2	2	1	OctaviuS
3	3	2	OctaviuS
4	4	3	OctaviuS
5	5	4	OctaviuS
6	6	5	OctaviuS
7	7	6	OctaviuS
8	8	7	OctaviuS
9	9	8	OctaviuS
10	10	9	OctaviuS
11	11	10	OctaviuS
12	12	11	yuppy
13	13	12	OctaviuS
14	14	13	OctaviuS
15	15	14	OctaviuS
16	16	15	OctaviuS
17	17	16	OctaviuS
18	18	17	OctaviuS
19	19	18	OctaviuS

З результатів виконання запитів один за одним, можемо побачити, що і в даному випадку тригер спрацював правильно. Це підтверджує таблиця “test”, а саме те, що спочатку у нашому запиті іде команда вставки(insert), результат якої оброблюється тригером та вносить необхідні дані у таблицю “test”.

Після цього виконується запит редагування(update). В цьому випадку також, результат оброблюється тригером та вносить необхідні дані до таблиці “test”.

Тобто, до таблиці “test” усі записи було додано у правильному порядку.

Завдання 4

READ COMMITTED

Read Committed — рівень ізоляції, який у Postgres є рівнем ізоляції за замовчуванням. Для транзакції на цьому рівні запит SELECT бачить ті дані, які були зафіксовані до початку запиту. SELECT ніколи не побачить незафіксованих даних або змін, внесених у процесі виконання запиту паралельними транзакціями. Поки працюють паралельні транзакції та поки вони не завершать своє виконання, змін у поточній транзакції видно не буде.

Дані після вставки, видалення та редагування у одній транзакції та інших:

The image displays four sequential screenshots of a PostgreSQL SQL Shell window, illustrating a series of database operations and queries under the Read Committed isolation level.

First Screenshot: Shows the initial setup and the first query. The user connects to the 'test' database. A table 'test' is created with columns 'id' (bigserial PRIMARY KEY), 'number' (bigint), and 'text' (text). Data is inserted into the table. The first query shows the state of the table before a transaction begins.

id	number	text
1	1111	value1
2	2222	value2
3	3333	value3
4	4444	value4

Second Screenshot: Shows the execution of a transaction. The user sets the transaction isolation level to 'read committed'. The second query shows the state of the table after the transaction begins. The data remains the same as in the first screenshot.

id	number	text
1	1111	value1
2	2222	value2
3	3333	value3
4	4444	value4

Third Screenshot: Shows the execution of a transaction. The user sets the transaction isolation level to 'read committed'. The third query shows the state of the table after the transaction begins. The data remains the same as in the first screenshot.

id	number	text
1	1111	value1
2	2222	value2
3	3333	value3
4	4444	value4

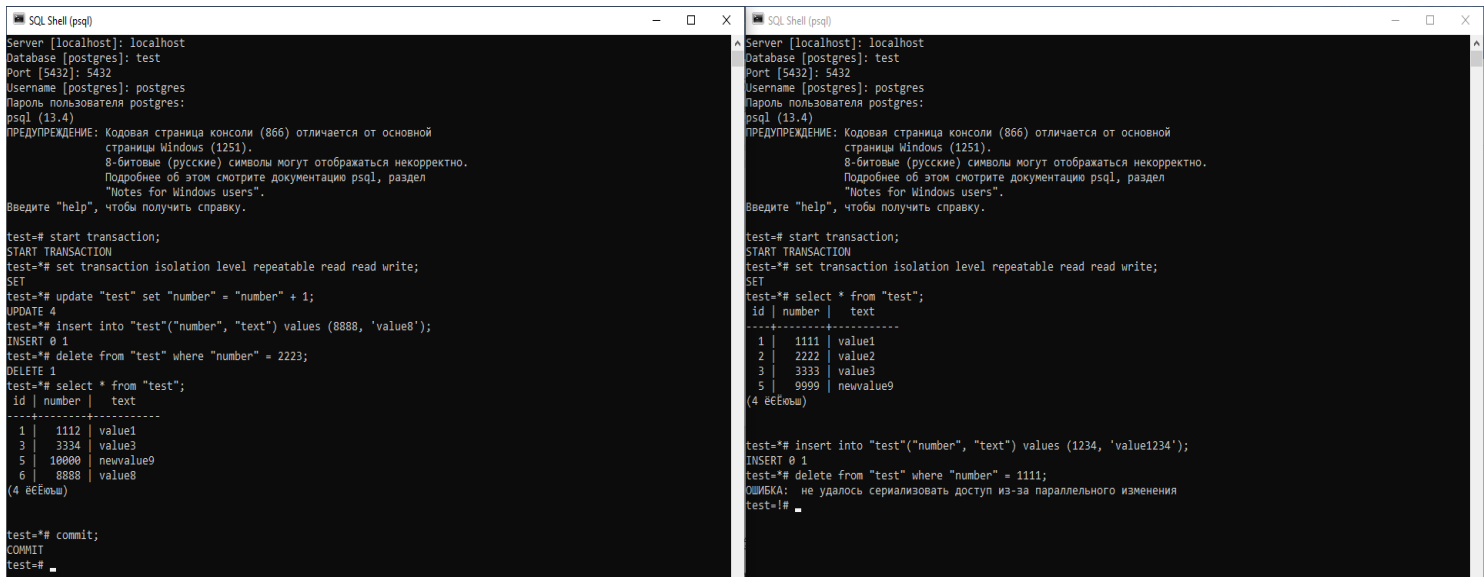
Fourth Screenshot: Shows the execution of a transaction. The user sets the transaction isolation level to 'read committed'. The fourth query shows the state of the table after the transaction begins. The data remains the same as in the first screenshot.

id	number	text
1	1111	value1
2	2222	value2
3	3333	value3
4	4444	value4

З результатів, отриманих на скріншотах, бачимо, що після встановлення рівню ізоляції та початку роботи у першій транзакції, друга транзакція не може вносити змін до таблиці “test” доти, доки перша транзакція не збереже змінені дані командою commit. Тобто, коли друга транзакція бачить зміни у першій транзакції за допомогою запитів UPDATE та DELETE, то виникає феномен повторного читання (транзакція, що читає, «не бачить» зміни даних, які були нею раніше прочитані), а при INSERT виникає читання фантомів (ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків). Тобто, на цьому рівні забезпечується захист від чорнового, «брудного» читання.

REPEATABLE READ

Repeatable Read – рівень ізоляції, при якому видно лише ті дані, які були зафіксовані до початку транзакції, але не видно незафіксовані дані та зміни, здійснені іншими транзакціями в процесі виконання цієї транзакції (однак запит бачитиме ефекти попередніх змін у своїй транзакції, незважаючи на те, що вони не зафіксовані).



```
SQL Shell (psql)
Server [localhost]: localhost
Database [postgres]: test
Port [5432]: 5432
Username [postgres]: postgres
Пароль пользователя postgres:
psql (13.4)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной
страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел
"Notes for Windows users".
Введите "help", чтобы получить справку.

test=# start transaction;
START TRANSACTION
test=# set transaction isolation level repeatable read read write;
SET
test=# update "test" set "number" = "number" + 1;
UPDATE 4
test=# insert into "test"("number", "text") values (8888, 'value8');
INSERT 0 1
test=# delete from "test" where "number" = 2223;
DELETE 1
test=# select * from "test";
 id | number | text
-----
  1 | 1112 | value1
  3 | 3334 | value3
  5 | 10000 | newvalue9
  6 | 8888 | value8
(4 строк)

test=# commit;
COMMIT
test=#
```

```
SQL Shell (psql)
Server [localhost]: localhost
Database [postgres]: test
Port [5432]: 5432
Username [postgres]: postgres
Пароль пользователя postgres:
psql (13.4)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной
страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел
"Notes for Windows users".
Введите "help", чтобы получить справку.

test=# start transaction;
START TRANSACTION
test=# set transaction isolation level repeatable read read write;
SET
test=# select * from "test";
 id | number | text
-----
  1 | 1111 | value1
  2 | 2222 | value2
  3 | 3333 | value3
  5 | 9999 | newvalue9
(4 строк)

test=# insert into "test"("number", "text") values (1234, 'value1234');
INSERT 0 1
test=# delete from "test" where "number" = 1111;
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
test=#
```

З даного скріншоту бачимо, що на даному рівню ізоляції виникає читання фантомів, тобто, insert може використовуватися у другій транзакції під час роботи першої. Але не все так просто: після виконання команди commit у першій транзакції, усі зміни, які відбулися у другій транзакції, можна сказати, зникнуть, і не будуть ніде відображені. На цьому рівні ізоляції є заборона іншим транзакціям змінювати рядки, які були зчитані незавершеною транзакцією. Однак, інші транзакції можуть вставляти нові рядки. Використання даного рівня ізоляції без необхідності не є доцільним.

SERIALIZABLE

Serializable – це найвищий рівень ізольованості. Транзакції повністю ізолюються одна від одної, кожна виконується так, ніби паралельних транзакцій не існує. Тільки на цьому рівні паралельні транзакції не схильні до ефекту “фантомного читання”.

```
SQL Shell (psql)
Server [localhost]: localhost
Database [postgres]: test
Port [5432]: 5432
Username [postgres]: postgres
Пароль пользователя postgres:
psql (13.4)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной
страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел
"Notes for Windows users".
Введите "help", чтобы получить справку.

test=# start transaction;
START TRANSACTION
test=# set transaction isolation level serializable read write;
SET
test=# update "test" set "number" = "number" / 2;
UPDATE 4
test=# insert into "test"("number", "text") values (5555, 'newvalue5');
INSERT 0 1
test=# delete from "test" where "id" = 1;
DELETE 1
test=# select * from "test";
 id | number | text
-----+-----+-----
  3 | 1667 | value3
  5 | 5000 | newvalue9
  6 | 4444 | value8
  8 | 5555 | newvalue5
(4 rows)

test=# commit;
COMMIT
test=#
```

```
SQL Shell (psql)
Server [localhost]: localhost
Database [postgres]: test
Port [5432]: 5432
Username [postgres]: postgres
Пароль пользователя postgres:
psql (13.4)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной
страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел
"Notes for Windows users".
Введите "help", чтобы получить справку.

test=# start transaction;
START TRANSACTION
test=# set transaction isolation level serializable read write;
SET
test=# select * from "test";
 id | number | text
-----+-----+-----
  1 | 1112 | value1
  3 | 3334 | value3
  5 | 10000 | newvalue9
  6 | 8888 | value8
(4 rows)

test=# update "test" set "number" = "number" + 5;
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
test=#
```

Даний рівень призначений для недопущення читання фантомів. На цьому рівні ізоляції гарантується максимальна узгодженість даних.